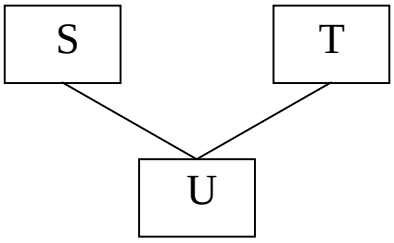
Multiple Inheritance Using Ada 95*

Martin C. Carlisle
Department of Computer Science
US Air Force Academy
mcc@cs.usafa.af.mil

Multiple inheritance is the idea that an object inherits attributes and methods from more than one parent, and the object itself is said to be a subclass of both parent classes. Difficulties in implementation cause many popular OO languages (e.g. Ada95 and Smalltalk) to avoid implementing multiple inheritance. For example, let's look at the following hierarchy:



If both S and T define a method "Bob" having the same arguments, what happens when "Bob" is invoked on an object of type U? Which, if either, of the methods from S and T are inherited? Despite such difficulties, we can accomplish almost all of the same objectives of multiple inheritance using a variety of mechanisms.

1. Combining generics and classes

Suppose we wish to combine the properties of an object class S with the properties of a linked list. Rather than having a class for a linked list object and the class S, we can instead implement the linked list package as a generic:

```
GENERIC
  TYPE ListObject IS ABSTRACT TAGGED PRIVATE;
PACKAGE ListPackage IS
  TYPE ListItem IS ABSTRACT NEW ListObject WITH PRIVATE;
```

---

* Appears in "Early Projects Using Ada at the United States Air Force Academy", Col. Samuel Grier, Proceedings of the 11th Annual ASEET Symposium, Monmouth University, June 1997.

```
  TYPE ListPointer IS ACCESS ALL ListItem'CLASS;

  PROCEDURE AddToList(List : IN OUT ListPointer;
    Object : IN ListObject);
  -- Define additional methods for a linked list here

PRIVATE
  TYPE ListItem IS NEW ListObject WITH RECORD
    Next : ListPointer;
  END RECORD;

END ListPackage;
```

In order to access the functionality of the linked list, we simply instantiate the package **ListPackage** using our class S as follows:

```
PACKAGE S_List IS NEW ListPackage(ListObject => S);
```

This is also referred to as "mixin inheritance" (Ada95 Rationale, Sec. 4.6.2). For this to work, one of the classes (in our example the list class) cannot have objects of its own, but merely exists to provide a set of properties for other classes.

There are, however, cases where each of the parents could have objects of its own, and the programmer wishes to have the child inherit from both parents. In these cases, we resort to one of two methods. In either case, the programmer will have to specify which of the two parents is the "real parent" (the other takes on more of a role of a "step parent"-- nothing is inherited, but it still provides for the child).

2. Declaring an object within another

Perhaps the simplest method of simulating multiple inheritance is to declare a child of one parent with an extension of the second parent. Using S,T, and U from our diagram, the type declarations would be as follows:

```
TYPE S IS TAGGED RECORD ... END RECORD;
PROCEDURE Method1(X : IN S);

TYPE T IS TAGGED RECORD ... END RECORD;
PROCEDURE Method1(X : IN T);
PROCEDURE Method2(X : IN T);

TYPE U IS NEW S WITH RECORD
  T_part : T;
END RECORD;

U1 : U;
```

By doing this, we have stated that S is the "real parent" of U. That is, U only inherits the methods from S directly—a call to **`Method1(U1)`** is valid and uses the method inherited from S; a call to **`Method2(U1)`** is not valid, rather the correct call would be **`Method2(U1.T_part)`**.

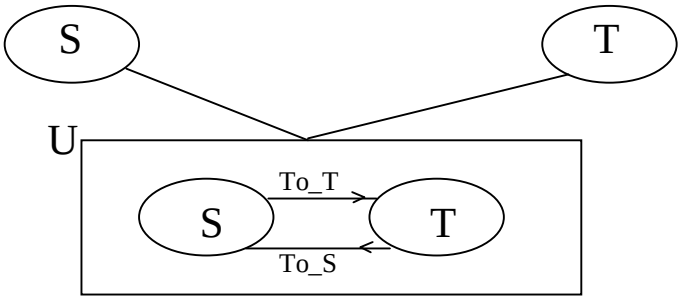
Membership tests are similarly restricted. **`U1 IN S'CLASS`** is true; however **`U1 IN T'CLASS`** is not. (In fact, if S and T are unrelated, i.e. they come from totally separate type hierarchies, the test **`U1 IN T'CLASS`** will generate a compile-time error.)

This also has the disadvantage that the view conversion is one-way. That is, once we "cast" U1 to type T by accessing the **`T_part`** field, we cannot "recast" back to type U or to type S. The last and most complicated mechanism addresses this concern.

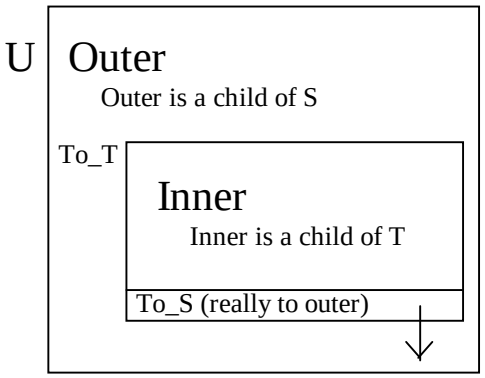
3. Multiple views

Multiple views (Ada95 Rationale, Sec. 4.6.3) allow us to create pointers from each representation of the object to the other representation by parameterizing one record with another. We still must declare one of the parents to be primary (this will be the parent of the *outer part*).

The actual implementation is illustrated by the following two diagrams. The first emphasizes that U consists of two objects, one of type S and one of type T, and that you can convert the view freely between these two views.



In the second diagram, we give a more realistic view of how U is implemented by the compiler. Actually, U will have an outer part, and an inner part. The outer part will be a child class of the "real parent", S, and will have inside it an object that is a child of type T. The inner part will point to the outer, and will be a field of the outer:



The declarations are as follows:

```
-- S and T must be limited types

   type Outer; -- child of S

-- declare inner type (child of T)
-- inherits all operations of T
   type Inner (To_S: access S'class)
        is new T with null record;


-- declare outer type
-- inherits all operations of S
   type Outer is new S with
      record
         To_T: aliased Inner (To_S => Outer'access);
      end record;

-- put methods here for the multiply inherited object.  For example, you
-- might want to override some methods of S to call methods from T
-- instead, or add new methods for this object

-- rename the outer part as U
   subtype U is Outer;
```

At this point, we'll examine what exactly is happening in the above declarations. The first line indicates that there will be a type called **`Outer`**, which will be defined later (this is needed for the **`Outer'access`** within the definition of **`Outer`**). We then declare the inner portion, i.e. the part that is a child of the "step parent". The declaration is parameterized with a pointer. This indicates that when we create an object of type **`Inner`**, we will give it a pointer to an object that is of type S or one of its descendants.

The next declaration is of the type **`Outer`**. Note that inside it is an object of type **`Inner`**. When this object is created, it is given a pointer to the outer part. Since this self-referring pointer

is present, we cannot copy the structure, hence we require the use of limited types (restricting the use of ":=" for assignment).

After the declaration of **Outer**, we can add any new methods or override any methods from S. Finally, we create a subtype to rename **Outer** as whatever we wish.

Note what we have gained over the method described in Section 2 is that we now have a way to go from the inner part of an object of type U (which is a child of T) to the outer part (containing both S and T). Consequently if we have variables **U1** of type **U** and **Inner1** of type **Inner**, we can use:

```
Inner1 := U1.To_T;
```
to get to the object descended from T and
```
Inner1.To_S;
```

to get back to the outer view. However, an object of type U is still not a member of **T'CLASS,** consequently as above **U1 IN T'CLASS** is false (and will generate a compile-time error if S and T are unrelated).